IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN XXXX-XXXX, Vol. 0, No. 0, pp. 1–26. DOI:XXXXXXXX

High-Throughput GPU Design and Implementation of MAYO with Matrix Computation Reordering

Haoyang Chen[1], Junhao Huang[2], Shutong Jin[3], Ray C. C. Cheung[3], Donglong Chen[4] and Wangchen Dai[1]*

[1] Sun Yat-sen University, Shenzhen, China
chenhy583@mail2.sysu.edu.cn, daiwch@mail.sysu.edu.cn
[2] Singapore Management University, Singapore
junhaohuang@smu.edu.sg
[3] City University of Hong Kong, Hong Kong, China
shutong.jin@my.cityu.edu.hk, r.cheung@cityu.edu.hk
[4] Beijing Normal-Hong Kong Baptist University, Zhuhai, China
donglongchen@bnbu.edu.cn

Abstract. With the rapid development of quantum computing, traditional digital signature schemes face increasing security threats. MAYO is a post-quantum digital signature scheme based on the hardness of the multivariate quadratic problem. Owing to its compact signature size, reduced public key, and flexible parameter sets, it has emerged as a competitive candidate in the NIST Additional Digital Signature round. However, existing MAYO implementations primarily target general-purpose processors, SIMD-enabled CPUs, or embedded platforms, and their throughput remains insufficient for high-demand scenarios such as server-side verification and large-scale authentication services. In this work, we present the first high-throughput GPU implementation and evaluation of MAYO, covering Keygen, Signing, and Verification. We introduce a data reuse-aware computation model for matrix-dominated and memory-access-bound workloads, and propose Reuse Matrix Multiplication Evaluation (RMME) to reduce redundant global memory accesses through operand-level data reuse. Building upon this method, we further develop a GPU-oriented hardware–software co-design framework that combines algorithmic restructuring, warp-level kernel design, task-level memory reuse, and asynchronous multi-stream execution to maximize end-to-end throughput. Across all four MAYO parameter sets, the proposed implementation achieves more than 700× higher Signing throughput and more than 600× higher Verification throughput than CPU-based implementations. In the best case, it supports millions of verifications per second. These results demonstrate that RMME, together with GPU-oriented hardware–software co-design, is highly effective for accelerating multivariate signature schemes and provides a practical high-throughput solution for post-quantum digital signatures.

Keywords: Post-quantum cryptography · Digital signatures · MAYO · GPU

1 Introduction

Digital signatures are cryptographic primitives for ensuring authenticity, integrity, and non-repudiation in modern digital systems. They are widely deployed in software distribution, secure communication, and blockchain systems. Today, most widely used signature

*Corresponding author.

schemes rely on classical number-theoretic assumptions such as integer factorization and discrete logarithms. With the rapid development of quantum computing, however, these assumptions are vulnerable to quantum algorithms [Sho99], rendering such signature schemes insecure, and motivating the development of post-quantum signature schemes.

In 2016, NIST launched the post-quantum cryptography standardization process and selected three post-quantum signature schemes for standardization in 2022 [AAC+22]. In the same year, to avoid over-reliance on a single mathematical problem, NIST initiated the Additional Post-Quantum Digital Signature Algorithm call. In 2024, NIST announced 14 algorithms that advanced to the second round. Among the candidates of the ongoing standardization process, multivariate signature schemes have attracted increasing attention due to their relatively compact signatures. In particular, MAYO, a recent multivariate signature scheme based on the hardness of the Multivariate Quadratic (MQ) problem [Beu21], has emerged as a competitive candidate in the NIST Additional Signature round. Compared to earlier Oil-and-Vinegar–type constructions [KPG99], MAYO introduces emulsifier maps and the whipping technique, achieving a significantly smaller public key while retaining a compact signature size.

Despite these advantages, the practical deployment of MAYO in high-throughput scenarios—such as large-scale authentication services and server-side verification—remains challenging. Existing implementations primarily target CPUs, SIMD (e.g., AVX2), or embedded processors. While these approaches achieve steady improvements, they are fundamentally constrained by limited parallelism and memory bandwidth, making them difficult to support the massive concurrency required in modern cloud-scale systems.

Graphics Processing Units (GPUs) offer massive parallelism and high memory bandwidth, and have been successfully applied to accelerate various cryptographic primitives, including lattice-based and hash-based post-quantum signature schemes. [ORCR20, LSKY19, FXL+22, SYD+24, LWS+26, WDC+25, SA23, SZM20, SYLZ25, LZS+24]. However, these techniques do not directly transfer to multivariate cryptography. In contrast to lattice-based schemes, which are typically compute-bound and benefit from arithmetic parallelism, multivariate schemes such as MAYO are dominated by large-scale structured matrix computations and exhibit memory-access-bound behavior. As a result, the primary performance bottleneck lies not in arithmetic throughput, but in inefficient global memory access patterns. Existing GPU optimization techniques, which largely focus on parallelizing independent computations or optimizing arithmetic kernels, fail to effectively exploit operand-level data reuse inherent in such workloads.

Related Work. Existing research on the MAYO signature scheme has primarily focused on implementations on general-purpose processors, embedded platforms, and dedicated hardware. Beullens et al. [Beu21] first introduced MAYO and provided a reference software implementation, laying the foundation for subsequent optimizations. Building upon this work, Beullens et al. [BCC+24] developed efficient implementations for AVX2-enabled processors and Arm Cortex-M4 platforms based on a nibble-sliced representation. Gringiani et al. [GMSS23] further optimized MAYO for Armv7-M embedded platforms by designing parameters suitable for 32-bit architectures and combining bitslicing with polar-form techniques. On the hardware side, Hirner et al. [HSK+24] proposed hardware implementations of MAYO, including high-speed ASIC and FPGA accelerators for the complete MAYO workflow. Despite these advances, the overall throughput of existing MAYO implementations remains insufficient for server-class scenarios.

In parallel, GPU-accelerated implementations of post-quantum digital signature schemes have made significant progress, but most existing efforts focus on lattice-based and hash-based constructions. For lattice-based signatures, Shen et al. [SYD+24] proposed a high-performance GPU implementation framework for Dilithium, achieving substantial performance improvements through thread-level parallel rejection sampling, memory pool management, and dynamic task scheduling. Li et al. [LWS+26] presented cuFalcon, an

adaptive parallel GPU implementation for the Falcon signature scheme, which attains a high throughput of approximately 200,000 signatures per second by leveraging FFT optimization, recursive sampling expansion, and kernel fusion. For hash-based signatures, Wang et al. [WDC+25] introduced the CUSPX framework to accelerate SPHINCS+ on GPU, reducing the latency of a single signature to 0.67 ms. These works demonstrate that GPU acceleration is highly effective for post-quantum signature schemes whose performance mainly benefits from arithmetic parallelism and optimized compute kernels.

Unlike lattice- and hash-based constructions, MAYO is fundamentally constrained by memory access rather than arithmetic throughput. Therefore, these optimization strategies cannot be directly applied to MAYO.

Our contributions. In this work, we introduce a data reuse-aware computation model for matrix-dominated and memory-access-bound workloads, and apply it to the design of a high-throughput GPU implementation of MAYO. Our approach is based on the key observation that many algorithms in MAYO, such as Keygen, Signing, and Verification, can be identified as sequences of matrix operations on shared operands. By explicitly modeling data reuse at the operand level, we derive an optimized evaluation strategy that minimizes redundant global memory accesses.

Concretely, we propose Reuse Matrix Multiplication Evaluation (RMME). The core idea of RMME is to reorganize matrix multiplication around a selected operand, such that each loaded element is immediately reused across all dependent multiply–accumulate operations before being discarded. This approach asymptotically reduces global memory access complexity and provides a principled foundation for optimizing memory-access-bound cryptographic workloads.

Building upon this method, we further develop a hardware–software co-design framework for GPU platforms. At the algorithmic level, we integrate RMME with phase fusion and algebraic restructuring (e.g., Horner-style evaluation) to eliminate intermediate storage and reduce repeated passes over large matrices. At the kernel level, we design warp-oriented execution strategies and lightweight cryptographic primitives to balance register usage and parallelism. At the system level, we employ task-level memory pooling and asynchronous multi-stream execution to overlap computation and data transfer, thereby improving end-to-end throughput.

We implement the proposed design on different GPU platforms and evaluate its performance across all MAYO parameter sets. Our results show that the proposed approach achieves orders-of-magnitude throughput improvements over existing CPU-based implementations, reaching millions of verifications per second in the best case. A detailed performance analysis reveals that our method effectively mitigates memory bottlenecks by significantly reducing global memory traffic, leading to high utilization of on-chip resources.

Our contributions are summarized as follows:

(1) A data reuse-aware computation model for multivariate cryptography. We formalize matrix-dominated computations in MAYO as memory-access-bound linear-algebraic operations and introduce RMME as a general framework for reducing redundant global memory accesses.

(2) Algorithmic restructuring of core MAYO computations. We systematically apply RMME and algebraic operations (including Horner-style evaluation) to Keygen, Signing, and Verification, reducing the number of passes over matrix groups in the computation of y from $\frac{k(k+1)}{2}$ to a single pass.

(3) A GPU-oriented hardware–software co-design framework. We develop a unified optimization framework spanning algorithmic, kernel-level, and system-level design to maximize GPU throughput.

(4) A high-throughput GPU implementation of MAYO. We present the first high-throughput GPU implementation of MAYO covering Keygen, Signing, and Verification across all parameter sets.

Code: https://github.com/SysuCrypto/cuMAYO

2 Background and Preliminaries

2.1 Notation

We denote by $\mathbb{F}_q$ the finite field with q elements. We denote by $\mathbb{B} = \{0, 1\}$ the binary alphabet, and by $\mathbb{B}^*$ the set of binary strings of arbitrary length. For a positive integer m, we write $[m] = \{0, 1, \ldots, m-1\}$. We write $a \leftarrow b$ to denote that a is assigned the value b. If x is a vector, $x[i]$ denotes its i-th element. If X is a matrix, $X[i, j]$ denotes the element in the i-th row and the j-th column. For a matrix X, $\mathsf{Upper}(X)$ denotes the upper triangular matrix obtained by adding each lower triangular entry $X[j, i]$ to the corresponding upper triangular entry $X[i, j]$ for $i < j$, and setting all entries below the diagonal to zero. If P represents a collection of m matrices, $P_a[i, j]$ denotes the element in the i-th row and the j-th column of the a-th matrix. The notation $||$ denotes the concatenation of two bit strings. The notation 0_n denotes the all-zero object of length n in the corresponding domain. Accordingly, $x \leftarrow 0_n$ means that x is initialized to zero with length n. The symbol $\perp$ denotes failure or the absence of a valid solution.

2.2 MAYO

MAYO is a post-quantum digital signature scheme based on multivariate polynomials and can be viewed as a variant of the Oil-and-Vinegar (UOV) scheme. Its security relies on the hardness of solving systems of multivariate quadratic equations over finite fields, known as the Multivariate Quadratic (MQ) problem. This problem is considered NP-hard in both classical and quantum settings.

The signature scheme consists of three procedures: Algorithm 1 for Keygen, Algorithm 2 for Signing, and Algorithm 3 for Verification. MAYO operates over the finite field $\mathbb{F}_{16}$; therefore, addition and subtraction are identical and are implemented as bitwise exclusive-or, denoted by $\oplus$. SHAKE256 denotes the extendable-output hash function specified in the SHA-3 standard, and AES-128-CTR is used as a pseudorandom generator to expand seeds into the required matrix coefficients. Algorithm 4 describes the Expandsk procedure for secret-key expansion, and Algorithm 5 describes the Expandpk procedure for public-key expansion. The notation E^ℓ denotes multiplication by z^ℓ modulo the polynomial $f(z)$, where $f(z)$ is the defining polynomial of the extension field representation of $\mathbb{F}_{16}$. The function SampleSolution denotes the procedure for solving the linear system $Ax = y$ over $\mathbb{F}_{16}$. The matrix P_a is constructed as

$$P_a = \begin{pmatrix} P_a^{(1)} & P_a^{(2)} \\ 0 & P_a^{(3)} \end{pmatrix}.$$

From a computational perspective, several core procedures in MAYO, including the construction of $P^{(3)}$ in Keygen, the construction of L and M in Signing, and the evaluation of the intermediate value y in Signing and Verification, involve matrix-related operations such as matrix–matrix and matrix–vector products. In reference implementations, these computations are typically evaluated using the conventional inner-product formulation. Since these matrix computations dominate the main arithmetic workload of MAYO, they become the primary target of our subsequent optimization.

MAYO defines four parameter sets: MAYO_1, MAYO_2, MAYO_3, and MAYO_5, targeting NIST security levels 1, 1, 3, and 5, respectively. Table 1 lists the parameters of these parameter sets.

Table 1: MAYO parameter sets

Parameter set	Level	n	m	o	k
MAYO_1	1	86	78	8	10
MAYO_2	1	81	64	17	4
MAYO_3	3	118	108	10	11
MAYO_5	5	154	142	12	12

Algorithm 1 MAYO.Keygen() [BCC+25]

Output: Public key $cpk \in \mathbb{B}^{\text{cpk\_bytes}}$
Output: Secret key $csk \in \mathbb{B}^{\text{csk\_bytes}}$

1: $seed_{sk} \leftarrow \mathbb{B}^{\text{sk\_seed\_bytes}}$
2: $(seed_{pk}, O) \leftarrow \mathsf{SHAKE256}(seed_{sk})$ $//O \in \mathbb{F}_q^{(n-o)\times o}, seed_{pk} \in \mathbb{B}^{pk\_seed\_bytes}$
3: $\{P_i^{(1)}, P_i^{(2)}\}_{i\in[m]} \leftarrow \mathsf{AES\text{-}128\text{-}CTR}(seed_{pk})$
$//P_i^{(1)} \in \mathbb{F}_q^{(n-o)\times(n-o)}$ *upper triangular*, $P_i^{(2)} \in \mathbb{F}_q^{(n-o)\times o}$
4: **for** $i = 0$ **to** $m-1$ **do**
5: $\quad P_i^{(3)} \leftarrow \mathsf{Upper}(-O^\top P_i^{(1)} O - O^\top P_i^{(2)})$ $//P_i^{(3)} \in \mathbb{F}_q^{o\times o}$ *upper triangular*
6: $cpk \leftarrow seed_{pk} \,\|\, \{P_i^{(3)}\}_{i\in[m]}$
7: $csk \leftarrow seed_{sk}$
8: **return** (cpk, csk)

2.3 GPU

In the CPU–GPU cooperative computing model, the CPU serves as the host and is responsible for program control and task scheduling, while the GPU acts as an accelerator for large-scale data-parallel computation. Computational tasks are launched from the CPU to the GPU in the form of kernels, each of which is executed by a large number of threads in parallel. CUDA further supports asynchronous execution through streams, allowing concurrent execution of multiple kernels as well as overlap between computation and data transfer.

On the GPU side, threads are organized into thread blocks, which are dynamically assigned to Streaming Multiprocessors for execution. Within each Streaming Multiprocessor (SM), threads are scheduled at the warp level, with each warp consisting of 32 threads, as shown in Figure 1. The GPU provides a hierarchical memory system to balance access latency and storage capacity, including registers, constant memory, shared memory, global memory, local memory, etc.

Registers reside within each SM and offer the lowest access latency, but their capacity is limited, making them suitable for storing thread-private temporary variables. Constant memory is a small, read-only memory space cached on-chip and optimized for broadcast access patterns, and is typically used to store frequently accessed constants shared across threads. Shared memory is also located on-chip within each SM and provides low-latency access, enabling efficient data reuse and cooperative computation within a thread block.

In contrast, global memory provides the largest storage capacity in the GPU memory hierarchy and is primarily used for storing large data structures shared across thread blocks. Accesses to global memory are serviced through a multi-level cache hierarchy. The L1 cache is located on-chip and closely associated with each SM, where it primarily serves to reduce access latency for frequently reused data. The L2 cache is a unified on-chip cache shared by all SMs, supporting data reuse across SMs and used to improve bandwidth utilization. At the lowest level of the memory hierarchy, global memory resides in off-chip DRAM, which offers high capacity but incurs the highest access latency. Local memory refers to per-thread storage used when variables cannot be allocated in registers. Such accesses

Algorithm 2 MAYO.Signing(csk, M) [BCC+25]

Input: Secret key $csk \in \mathbb{B}^{\text{csk\_bytes}}$.
Input: Message $M \in \mathbb{B}^*$.
Output: Signature sig $\in \mathbb{B}^{\text{sig\_bytes}}$.

$\{seed_{sk}, O, \{P_i^{(1)}\}_{i\in[m]}, \{L_i\}_{i\in[m]}\} \leftarrow \mathsf{Expandsk}(csk)$ // $L_i \in \mathbb{F}_q^{(n-o)\times o}$
$M_{\text{digest}} \leftarrow \mathsf{SHAKE256}(M,\ \text{digest\_bytes})$ // $M_{digest} \in \mathbb{B}^{digest\_bytes}$
$R \leftarrow 0_{\text{R\_bytes}}$ **or** $R \leftarrow \mathbb{B}^{\text{R\_bytes}}$
$salt \leftarrow \mathsf{SHAKE256}(M_{\text{digest}} \,\|\, R \,\|\, seed_{sk})$ // $salt \in \mathbb{B}^{salt\_bytes}$
$t \leftarrow \mathsf{SHAKE256}(M_{\text{digest}} \,\|\, salt)$ // $t \in \mathbb{F}_q^m$
for $ctr = 0$ to 255 **do**
  $V \leftarrow \mathsf{SHAKE256}(M_{\text{digest}} \,\|\, salt \,\|\, seed_{sk} \,\|\, ctr)$ // $V \in \mathbb{F}_q^{k\times(n-o)+\lceil ko\log(q)/8\rceil}$
  for $i = 0$ to $k-1$ **do**
    $v_i \leftarrow V[i\cdot\text{v\_bytes} : (i+1)\cdot\text{v\_bytes}]$
  $r \leftarrow V[k\cdot\text{v\_bytes} : k\cdot\text{v\_bytes} + \lceil ko\log(q)/8\rceil]$
  for $i = 0$ to $k-1$ **do**
    for $j = 0$ to $m-1$ **do**
      $M_i[j,:] \leftarrow v_i^\top L_j$ // $M_i \in \mathbb{F}_q^{m\times o}$
  $\ell \leftarrow 0$
  for $i = 0$ to $k-1$ **do**
    for $j = k-1$ to i **do**
      $A[:,\, i\cdot o : (i+1)\cdot o] \leftarrow A[:,\, i\cdot o : (i+1)\cdot o] + E^\ell M_j$ // $A \in \mathbb{F}_q^{m\times ko}$
      if $i \neq j$ **then**
        $A[:,\, j\cdot o : (j+1)\cdot o] \leftarrow A[:,\, j\cdot o : (j+1)\cdot o] + E^\ell M_i$
      $\ell \leftarrow \ell + 1$
  $y \leftarrow t, \ell \leftarrow 0$
  for $i = 0$ to $k-1$ **do**
    for $j = k-1$ to i **do**
      $$u \leftarrow \begin{cases} \{v_i^\top P_a^{(1)} v_i\}_{a\in[m]} & \text{if } i = j, \\ \{v_i^\top P_a^{(1)} v_j + v_j^\top P_a^{(1)} v_i\}_{a\in[m]} & \text{if } i \neq j \end{cases}$$
      $y \leftarrow y - E^\ell u$
      $\ell \leftarrow \ell + 1$
  $x \leftarrow \mathsf{SampleSolution}(A, y, r)$ // $x \in \mathbb{F}_q^{ko} \cup \{\perp\}$
  if $x \neq \perp$ **then**
    break
for $i = 0$ to $k-1$ **do**
  $s[i\cdot n : (i+1)\cdot n] \leftarrow (v_i + O\,x[i\cdot o : (i+1)\cdot o]) \,\|\, x[i\cdot o : (i+1)\cdot o]$ // $s \in \mathbb{F}_q^{kn}$
return sig $= s \,\|\, salt$

are mapped to the global memory address space and are therefore serviced through the same memory hierarchy, typically incurring latency comparable to that of global memory accesses. Despite the presence of caching mechanisms, accesses to global and local memory remain significantly more expensive than accesses to registers and shared memory. The effective bandwidth of shared memory is often about 12×–22× higher than that of global memory [JMSS18]. Therefore, efficient GPU implementations should maximize data reuse in shared memory whenever possible and reduce accesses to global memory, which also motivates the optimization strategies developed in this work.

Algorithm 3 MAYO.Verification(cpk, M, sig) [BCC+25]

Input: Public key $cpk \in \mathbb{B}^{\text{cpk\_bytes}}$
Input: Message $M \in \mathbb{B}^*$
Input: Signature sig $\in \mathbb{B}^{\text{sig\_bytes}}$
Output: An integer result to indicate if sig is valid (result = 0) or invalid (result = 1)

$\{P_i^{(1)}, P_i^{(2)}, P_i^{(3)}\}_{i\in[m]} \leftarrow \mathsf{Expandpk}(cpk)$
$salt \leftarrow sig[nk : nk + salt\_\text{bytes}]$
for $i = 0$ to $k - 1$ **do**
    $s_i \leftarrow sig[i \cdot n : (i + 1) \cdot n]$
$M_{\text{digest}} \leftarrow \mathsf{SHAKE256}(M)$
$t \leftarrow \mathsf{SHAKE256}(M_{\text{digest}} \,\|\, salt)$ //$t \in \mathbb{F}_q^m$
$y \leftarrow 0_m, \ell \leftarrow 0$
for $i = 0$ to $k - 1$ **do**
    for $j = k - 1$ to i **do**
        $u \leftarrow \begin{cases} \left\{ s_i^\top P_a s_i \right\}_{a\in[m]} & \text{if } i = j \\ \left\{ s_i^\top P_a s_j + s_j^\top P_a s_i \right\}_{a\in[m]} & \text{if } i \neq j \end{cases}$ //$u \in \mathbb{F}_q^m$
        $y \leftarrow y + E^\ell u$
        $\ell \leftarrow \ell + 1$
if $y = t$ **then**
    return 0
return 1

Algorithm 4 MAYO.Expandsk(csk) [BCC+25]

Input: Secret key $csk \in \mathbb{B}^{\text{csk\_bytes}}$
Output: Expanded secret key esk

$\{seed_{pk}, O\} \leftarrow \mathsf{SHAKE256}(csk)$
$\{P_i^{(1)}, P_i^{(2)}\}_{i\in[m]} \leftarrow \mathsf{AES\text{-}128\text{-}CTR}(seed_{pk})$
for $i = 0$ to $m - 1$ **do**
    $L_i \leftarrow \left(P_i^{(1)} + (P_i^{(1)})^\top\right) O + P_i^{(2)}$
return $esk \leftarrow csk \,\|\, O \,\|\, \{P_i^{(1)}\}_{i\in[m]} \,\|\, \{L_i\}_{i\in[m]}$

3 GPU-Oriented Design of Matrix Computations

3.1 Reuse Matrix Multiplication Evaluation

To improve throughput, we exploit instance-level parallelism by scheduling multiple MAYO instances to execute concurrently on the same SM, while constraining per-instance resource usage. Due to strong intra-instance data dependencies, we adopt a warp-level execution model and use a thread-per-matmul mapping, in which each thread processes a group of complete matrix multiplication tasks sequentially.

Within a single MAYO instance, many core computations involve sequences of related matrix multiplications on shared operands. A representative form can be written as

$$C_i = A_i B, \qquad i = 0, 1, \ldots, t - 1,$$

where $\{A_i\}_{i\in[t]}$ denotes a collection of matrices and B is a shared matrix operand. Similar patterns also arise when the shared operand appears on the left-hand side. For example, in Keygen, the computation of $P_i^{(1)}O$ for all $i \in [m]$ follows this pattern, where $\{P_i^{(1)}\}_{i\in[m]}$ denotes a collection of matrices and O is shared across them. Similar shared-operand

Algorithm 5 MAYO.Expandpk(cpk) [BCC+25]

Input: Public key $cpk \in \mathbb{B}^{\text{cpk\_bytes}}$
Output: Expanded public key epk

1: $\{seed_{pk}, \{P_i^{(3)}\}_{i\in[m]}\} \leftarrow cpk$
2: $epk \leftarrow \text{AES-128-CTR}(seed_{pk}) \,\|\, \{P_i^{(3)}\}_{i\in[m]}$
3: **return** epk

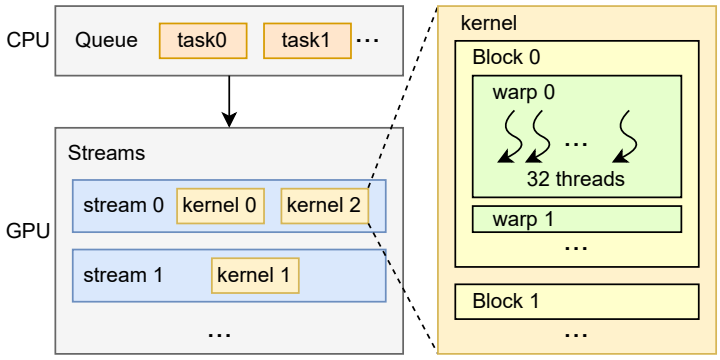


Figure 1: Overview of the CUDA execution workflow, illustrating CPU task dispatch, GPU streams, and the hierarchical organization of kernels, thread blocks, warps, and threads.

patterns also appear in the construction of L and in the evaluation of bilinear terms in Signing and Verification. Under the thread-per-matmul-task mapping, a straightforward implementation typically adopts the conventional inner-product formulation. For a matrix multiplication $C = AB$, each output element is computed independently as

$$C[m,n] = \sum_{k=0}^{K-1} A[m,k] \cdot B[k,n].$$

This evaluation order traverses output positions (m, n) and performs an independent inner product for each output element. As a result, the same matrix elements are repeatedly loaded across different output positions during the accumulation process. More specifically, under the conventional inner-product formulation, each element $A[m, k]$ is reused across all N output columns in row m, while each element $B[k, n]$ is reused across all M output rows in column n. When the input matrices reside in global memory, such repeated accesses create a severe global-memory-access bottleneck and significantly limit the overall throughput.

Building on this data reuse-aware computation model for matrix-dominated and memory-access-bound workloads, we propose Reuse Matrix Multiplication Evaluation (RMME), a computation reordering method for matrix multiplication. The core idea of RMME is to reorganize the computation around a selected input operand. During the evaluation, one element from the selected operand is loaded and immediately reused in all multiply-accumulate operations to which it contributes, while the corresponding partial sums are maintained in a set of intermediate variables. Only after all contributions associated with the current loaded element have been consumed does the computation proceed to the next element, as illustrated in Figure 2. In this way, RMME preserves the mathematical result of matrix multiplication while changing the order in which input elements are consumed and accumulated, thereby improving intra-thread data reuse and reducing redundant accesses to global memory.

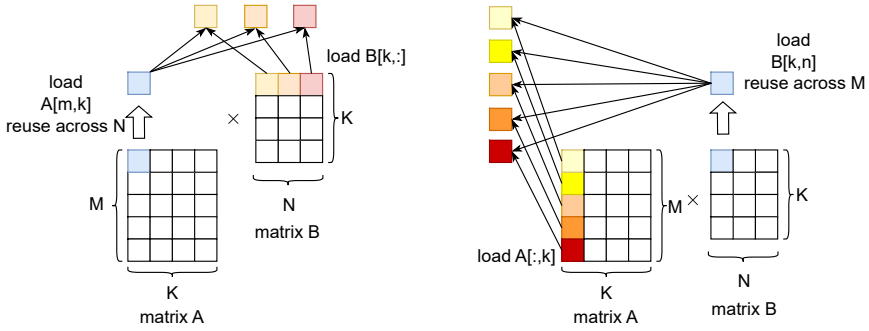


Figure 2: Illustration of RMME. Left: one element $A[m,k]$ is loaded and multiplied with all elements in row k of B before being discarded. Right: one element $B[k,n]$ is loaded and multiplied with all elements in column k of A before being discarded.

Algorithm 6 A-RMME: Reuse Left Matrix Multiplication Evaluation

Input: Matrix $A \in \mathbb{F}_q^{M\times K}$, matrix $B \in \mathbb{F}_q^{K\times N}$
Output: Matrix product $C = AB \in \mathbb{F}_q^{M\times N}$

for $m = 0$ to $M-1$ **do**
  $tmp \leftarrow 0_N$ *//$tmp \in \mathbb{F}_q^N$ stores the m-th row result of $A \times B$*
  for $k = 0$ to $K-1$ **do**
    $a \leftarrow A[m,k]$
    for $n = 0$ to $N-1$ **do**
      $tmp[n] \leftarrow tmp[n] + a \cdot B[k,n]$
  $C[m,:] \leftarrow tmp$

Depending on which operand is selected for data reuse, RMME admits two symmetric variants. When the left matrix is selected, the computation is organized around the rows of A. For each row index m, the algorithm loads one element $A[m,k]$ at a time and reuses it across all output columns associated with that row. The partial sums of the m-th output row are accumulated in an intermediate vector, as shown in Algorithm 6. Conversely, when the right matrix is selected, the computation is organized around the columns of B. For each column index n, the algorithm loads one element $B[k,n]$ at a time and reuses it across all output rows associated with that column. In this case, the partial sums of the n-th output column are accumulated in an intermediate vector, as shown in Algorithm 7. Therefore, A-RMME and B-RMME follow the same data reuse-driven principle, but differ in which operand is explicitly reused and in whether the intermediate result is maintained row-wise or column-wise.

For a representative matrix multiplication $C = AB$, where $A \in \mathbb{F}^{M\times K}$ and $B \in \mathbb{F}^{K\times N}$, Table 2 compares the memory read counts and register requirements of three single-thread evaluation strategies: the baseline inner-product formulation, A-RMME, and B-RMME. In the baseline inner-product formulation, each output element is computed as an independent length-K dot product, requiring K reads from A and K reads from B. Across all $M \cdot N$ output elements, this results in $M \cdot N \cdot K$ reads from both A and B. Only one register is required for accumulation.

When reusing matrix A (A-RMME), each element of A is loaded once and reused across all N output columns, reducing the total read count of A from $M \cdot N \cdot K$ to $M \cdot K$ and the per-element read complexity from $\mathcal{O}(N)$ to $\mathcal{O}(1)$. The reads from B remain unchanged. This optimization requires N accumulation registers for the output columns and one additional register for the loaded matrix element.

Algorithm 7 B-RMME: Reuse Right Matrix Multiplication Evaluation

Input: Matrix $A \in \mathbb{F}_q^{M \times K}$, matrix $B \in \mathbb{F}_q^{K \times N}$
Output: Matrix product $C = AB \in \mathbb{F}_q^{M \times N}$

1: **for** $n = 0$ to $N - 1$ **do**
2: $\quad tmp \leftarrow 0_M$ $\quad$ *//$tmp \in \mathbb{F}_q^M$ stores the n-th column result of $A \times B$*
3: $\quad$ **for** $k = 0$ to $K - 1$ **do**
4: $\quad\quad b \leftarrow B[k, n]$
5: $\quad\quad$ **for** $m = 0$ to $M - 1$ **do**
6: $\quad\quad\quad tmp[m] \leftarrow tmp[m] + b \cdot A[m, k]$
7: $\quad C[:, n] \leftarrow tmp$

Table 2: Memory access counts for single-thread matrix multiplication.

Method	A Reads	B Reads	Registers
Baseline (inner product)	$M \cdot N \cdot K$	$M \cdot N \cdot K$	1
A-RMME	$M \cdot K$	$M \cdot N \cdot K$	$N + 1$
B-RMME	$M \cdot N \cdot K$	$N \cdot K$	$M + 1$

Similarly, when reusing matrix B (B-RMME), each element of B is loaded once and reused across all M output rows, reducing the total read count of B from $M \cdot N \cdot K$ to $N \cdot K$ and the per-element read complexity from $\mathcal{O}(M)$ to $\mathcal{O}(1)$. The reads from A remain unchanged. This optimization requires M accumulation registers and one additional register for the loaded matrix element.

The choice between A-RMME and B-RMME depends on three considerations. First, the operand with the higher global-memory read cost should be prioritized for reuse. Therefore, A-RMME is preferred when reading A incurs a higher cost, whereas B-RMME is preferred when reading B incurs a higher cost. Second, the available register budget should be considered, since A-RMME requires N + 1 registers per thread, whereas B-RMME requires M + 1; the selected variant should avoid excessive register pressure and register spilling. Third, when the memory cost and register requirements of the two variants are comparable, the reuse factor is considered. Specifically, A-RMME reuses each loaded element of A across N output columns, whereas B-RMME reuses each loaded element of B across M output rows. The variant with the higher reuse factor is generally preferred.

Applicability to Other Multivariate Schemes. RMME is applicable to multivariate post-quantum signature implementations involving repeated matrix multiplication operations with shared operands. Taking the key-generation phase as an example, RMME can be applied to the computation of $\bar{P}_{i,3}$ in QR-UOV [FIH+23], the computation of P_i^{22} in SNOVA [WCD+25], and the computation of $P_i^{(3)}$ in UOV [BCD+25]. The performance gain depends on the proportion of such computations in the overall implementation and the available register budget.

Applicability to AVX2 Implementations. Although this work focuses on GPU implementations, the reuse principle of RMME can also be applied to AVX2 implementations. However, repeated memory accesses on CPUs are more effectively mitigated by cache locality, and existing AVX2 implementations of MAYO already exploit SIMD registers efficiently [BCC+24]. As a result, RMME may still benefit AVX2 implementations with substantial matrix computations, but the throughput improvement is expected to be smaller than on GPUs.

3.2 RMME-Based and Phase-Fused Evaluation of $P^{(1)}$ in Keygen

In the Keygen phase, the construction of $P_i^{(3)}$ is given in lines 4–5 of Algorithm 1:

$$P_i^{(3)} = \mathsf{Upper}\big(-O^\top P_i^{(1)} O - O^\top P_i^{(2)}\big).$$

By factoring out the common left multiplier $O^\top$, this expression can be rewritten as

$$P_i^{(3)} = \mathsf{Upper}\Big(O^\top \big(P_i^{(1)} O + P_i^{(2)}\big)\Big),$$

which reduces the number of matrix multiplications from three to two.

Based on this reformulation, a straightforward two-phase computation first constructs the intermediate matrix $B_i = P_i^{(1)} O + P_i^{(2)}$, and then computes $P_i^{(3)} = \mathsf{Upper}(O^\top B_i)$. However, this baseline organization has two drawbacks. First, it requires materializing the full intermediate matrix B_i, which introduces additional storage overhead. Second, in the computation of $P_i^{(1)} O$, the matrix $P_i^{(1)}$ is repeatedly accessed across different output columns, leading to redundant global memory accesses.

To address this issue, we apply RMME to the computation of $P_i^{(1)} O$. Since $P_i^{(1)}$ resides in global memory whereas O is cached in shared memory, we adopt the variant that explicitly reuses elements of $P_i^{(1)}$. Under this formulation, the computation of $P_i^{(1)} O$ is naturally organized row by row. For each row index r, RMME produces the r-th row of the product by reusing the loaded elements of the r-th row of $P_i^{(1)}$ across all output columns.

Based on this row-wise organization, we further fuse the subsequent computation at the row granularity, as summarized in Algorithm 8. Once the r-th row of $P_i^{(1)} O$ is obtained, it is immediately added to the r-th row of $P_i^{(2)}$ to form the corresponding row of B_i. This row is then used together with the r-th row of O to directly update the upper-triangular entries of $P_i^{(3)}$. More specifically, for each pair (i, j) with $i \le j$, the contribution of the current row is accumulated into $P_i^{(3)}[i, j]$ using $B_{\mathrm{row}}[j] \cdot O[r, i]$ and, when $i \neq j$, the symmetric term $B_{\mathrm{row}}[i] \cdot O[r, j]$. Therefore, the full intermediate matrix B_i no longer needs to be explicitly stored in memory; only the currently processed row is retained, as illustrated in Figure 3. At the implementation level, the accumulations for $P_i^{(3)}$ are maintained in registers and written back to memory only after all row contributions have been completed.

3.3 RMME and Horner-Based Evaluation of y in Signing

In the Signing phase, the computation of the intermediate value y is given in lines 21–26 of Algorithm 2. In the baseline workflow, y is constructed incrementally using the index pair (i, j) as the outer loop. For each (i, j), an intermediate vector $u_{i,j} \in \mathbb{F}^m$ is first computed, and then y is updated using the recurrence exponent ℓ, followed by incrementing ℓ. In total, $\frac{k(k+1)}{2}$ updates are performed. This (i, j)-major evaluation order repeatedly accesses the same matrix elements across different (i, j) combinations, thereby incurring substantial redundant global memory accesses and becoming a major performance bottleneck.

To address this issue, we restructure the computation of y into two parts. First, since the update of y is a linear accumulation over mutually independent $u_{i,j}$ terms, the generation of $u_{i,j}$ can be decoupled from the recurrence update of y. Second, the original recurrence involving the exponent ℓ can be rewritten into a Horner-style nested accumulation. Horner's method [BFST03] is a classical technique for efficient polynomial evaluation, whose core idea is to evaluate a polynomial as a sequence of nested multiply–add operations. For a polynomial

$$p(x) = a_0 + a_1 x + \cdots + a_n x^n,$$

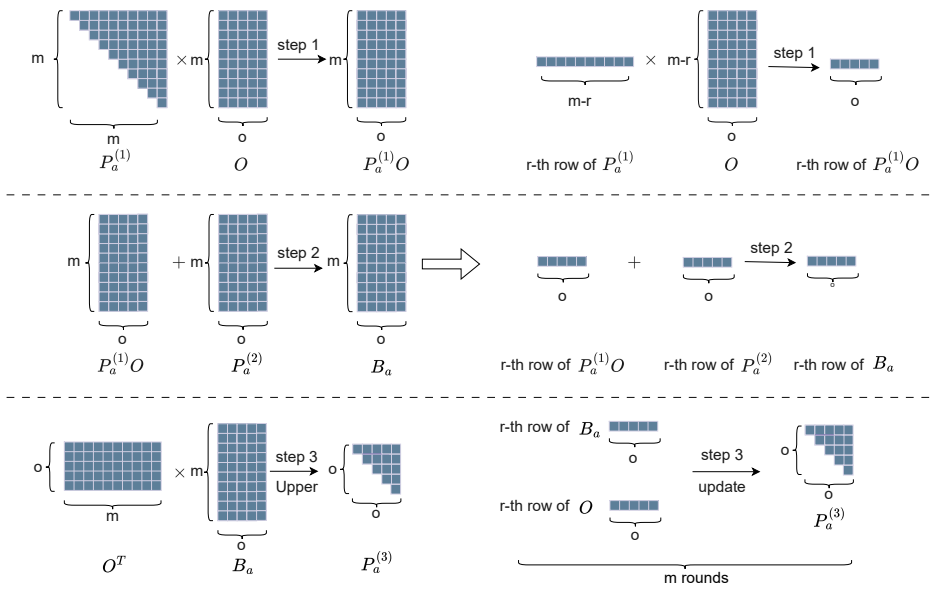


Figure 3: Comparison between the baseline construction with explicit intermediate matrices and the phase-fused evaluation without intermediate materialization.

the same expression can be rewritten as

$$p(x) = (\cdots((a_n x + a_{n-1})x + a_{n-2})x + \cdots + a_0).$$

This form avoids explicitly computing the powers of x and evaluates the recurrence in a nested manner. In our setting, the update of y can be reorganized in the same spirit, so that the accumulation is performed from back to front without explicitly handling each power term.

To efficiently generate $u_{i,j}$, we observe that the underlying computation can be reformulated as a matrix–matrix–matrix multiplication. For a matrix A and a sequence of column vectors $(b_0, b_1, \ldots, b_k)$, we have

$$\Big[Ab_0,\ Ab_1,\ \ldots,\ Ab_k\Big] = A\Big[b_0,\ b_1,\ \ldots,\ b_k\Big] = AB,$$

where B is formed by stacking these vectors column-wise. Similarly, for a sequence of row vectors $(c_0^\top, c_1^\top, \ldots, c_k^\top)$, we have

$$\begin{bmatrix} c_0^\top A \\ c_1^\top A \\ \vdots \\ c_k^\top A \end{bmatrix} = \begin{bmatrix} c_0^\top \\ c_1^\top \\ \vdots \\ c_k^\top \end{bmatrix} A = CA,$$

where C is the matrix formed by stacking these vectors row-wise. Accordingly, the computation of $u_{i,j}$, which involves multiplying one sequence of vectors by a matrix and then by another sequence of vectors, can be reformulated as a matrix–matrix–matrix multiplication.

Based on this reformulation, we apply RMME to the evaluation of $u_{i,j}$. Since the matrices in $P^{(1)}$ reside in global memory while the sequences of vectors are cached in shared memory, we adopt the RMME variant that explicitly reuses elements of $P^{(1)}$. The computation proceeds row by row over $P_a^{(1)}$. Once RMME produces the intermediate result

Algorithm 8 Computation of $P_a^{(3)}$

Input: Matrix $P_a^{(1)} \in \mathbb{F}_q^{m \times m}$ upper triangular
Input: Matrix $P_a^{(2)} \in \mathbb{F}_q^{m \times o}$
Input: Matrix $O \in \mathbb{F}_q^{m \times o}$
Output: Matrix $P_a^{(3)} \in \mathbb{F}_q^{o \times o}$ upper triangular

1: **for** $r = 0$ to $m-1$ **do**
2: $\quad B_{row} \leftarrow 0_o$ $\quad //B_{row} \in \mathbb{F}_q^o$
3: $\quad$ **for** $c = r$ to $m-1$ **do**
4: $\quad\quad p \leftarrow P_a^{(1)}[r, c]$
5: $\quad\quad$ **for** $k = 0$ to $o-1$ **do**
6: $\quad\quad\quad B_{row}[k] \leftarrow B_{row}[k] \oplus p \cdot O[c, k]$
7: $\quad$ **for** $k = 0$ to $o-1$ **do**
8: $\quad\quad B_{row}[k] \leftarrow B_{row}[k] \oplus P_a^{(2)}[r, k]$
9: $\quad$ **for** $i = 0$ to $o-1$ **do**
10: $\quad\quad$ **for** $j = i$ to $o-1$ **do**
11: $\quad\quad\quad P_a^{(3)}[i, j] \leftarrow P_a^{(3)}[i, j] \oplus B_{row}[j] \cdot O[r, i]$
12: $\quad\quad\quad$ **if** $i \neq j$ **then**
13: $\quad\quad\quad\quad P_a^{(3)}[i, j] \leftarrow P_a^{(3)}[i, j] \oplus B_{row}[i] \cdot O[r, j]$

associated with the r-th row, this row-wise intermediate result is immediately multiplied with the corresponding elements $v_i[r]$ and $v_j[r]$, and the resulting products are accumulated into the corresponding entries of $u_{i,j}$.

Based on the above reformulation, the RMME-based generation of $u_{i,j}$ and the Horner-style accumulation of y can be integrated into a single workflow, as summarized in Algorithm 9. This design significantly reduces global memory traffic. In particular, the number of passes over memory for $P_a^{(1)}$ is reduced from $\frac{k(k+1)}{2}$ in the baseline formulation to a single pass, providing a key performance advantage for the GPU implementation.

Correctness. Algorithm 9 differs from lines 21–26 of Algorithm 2 only in the evaluation order. Specifically, the implicit iteration over the matrix collection $\{P_a^{(1)}\}_{a \in [m]}$ in line 24 of Algorithm 2 is made explicit and reorganized using RMME in lines 3–14 of Algorithm 9, and the accumulation with powers of E in lines 25–26 of Algorithm 2 is rewritten in Horner form in lines 15–17 of Algorithm 9. Since MAYO is defined over a characteristic-two field, subtraction is equivalent to addition; for instance, $y - E^\ell u$ is equivalent to $y \oplus E^\ell u$. It remains to show that **(i)** lines 15–17 of Algorithm 9 compute the same E-weighted accumulation as lines 22–23 and 25–26 of Algorithm 2, and **(ii)** lines 3–14 of Algorithm 9 compute the same $u[i, j]$ as line 24 of Algorithm 2 for all pairs generated by lines 22–23.

(i) By unrolling the iteration, the computation of y can be written as

$$\begin{aligned} y = t &\oplus u[0, k-1] \oplus Eu[0, k-2] \oplus \cdots \oplus E^{k-1}u[0, 0] \\ &\oplus E^k u[1, k-1] \oplus \cdots \oplus E^{2k-2}u[1, 1] \\ &\oplus \cdots \\ &\oplus E^{\frac{k(k+1)}{2}-1}u[k-1, k-1]. \end{aligned}$$

Applying the Horner method to the above equation gives

$$y = t \oplus \Big(u[0, k-1] \oplus E\big(u[0, k-2] \oplus \cdots \oplus E(u[k-1, k-1])\big)\Big),$$

where $u[i, j]$ denotes the intermediate vector computed for the pair (i, j). Thus, lines 15–17

Algorithm 9 RMME and Horner-Based Evaluation of y in Signing

Input: Collection of matrices $P^{(1)} = \{P_a^{(1)}\}_{a\in[m]}$, where $P_a^{(1)} \in \mathbb{F}_q^{m\times m}$ upper triangular
Input: Collection of vectors $V = \{v_i\}_{i\in[k]}$, where $v_i \in \mathbb{F}_q^m$
Input: $t \in \mathbb{F}_q^m$
Output: $y \in \mathbb{F}_q^m$

1: $u[i,j] \leftarrow 0_m$, $0 \le i \le j < k$ //$u[i,j] \in \mathbb{F}_q^m$
2: $y \leftarrow 0_m$ //$y \in \mathbb{F}_q^m$
3: **for** $a = 0$ to $m-1$ **do**
4: **for** $r = 0$ to $m-1$ **do**
5: $T \leftarrow 0_k$ //$T \in \mathbb{F}_q^k$
6: **for** $c = r$ to $m-1$ **do**
7: $p \leftarrow P_a^{(1)}[r,c]$
8: **for** $i = 0$ to $k-1$ **do**
9: $T[i] \leftarrow T[i] \oplus (p \cdot v_i[c])$
10: **for** $i = 0$ to $k-1$ **do**
11: **for** $j = i$ to $k-1$ **do**
12: $u[i,j][a] \leftarrow u[i,j][a] \oplus (T[j] \cdot v_i[r])$
13: **if** $i \neq j$ **then**
14: $u[i,j][a] \leftarrow u[i,j][a] \oplus (T[i] \cdot v_j[r])$
15: **for** $i = k-1$ down to 0 **do**
16: **for** $j = i$ to $k-1$ **do**
17: $y \leftarrow Ey \oplus u[i,j]$
18: $y \leftarrow y \oplus t$

of Algorithm 9 compute the Horner accumulation, and line 18 adds t, giving the same E-weighted accumulation as lines 22–23 and 25–26 of Algorithm 2.

(ii) For a fixed matrix index a and row index r, lines 6–9 of Algorithm 9 load the entries of the r-th row of $P_a^{(1)}$ element by element. Since $P_a^{(1)}$ is upper triangular, only elements with $c \ge r$ are loaded. After the loop over c completes, for any vector index $i \in [k]$, the temporary vector satisfies

$$T[i] = \sum_{c=r}^{m-1} P_a^{(1)}[r,c] \cdot v_i[c] = (P_a^{(1)} v_i)[r].$$

When $i = j$, substituting the expression of $T[i]$ and accumulating $T[i] \cdot v_i[r]$ over all rows yields

$$u[i,i][a] = \sum_{r=0}^{m-1} v_i[r] \sum_{c=r}^{m-1} P_a^{(1)}[r,c] v_i[c] = v_i^\top P_a^{(1)} v_i.$$

When $i \neq j$, the row-wise accumulation term is $T[j] \cdot v_i[r] + T[i] \cdot v_j[r]$. Substituting the expressions of $T[j]$ and $T[i]$ and accumulating these terms over all rows yields

$$\begin{aligned} u[i,j][a] &= \sum_{r=0}^{m-1} v_i[r] \sum_{c=r}^{m-1} P_a^{(1)}[r,c] v_j[c] + \sum_{r=0}^{m-1} v_j[r] \sum_{c=r}^{m-1} P_a^{(1)}[r,c] v_i[c] \\ &= v_i^\top P_a^{(1)} v_j + v_j^\top P_a^{(1)} v_i. \end{aligned}$$

Thus, for every $a \in [m]$ and every $0 \le i \le j < k$, Algorithm 9 computes exactly the same $u[i,j][a]$ as in Algorithm 2. As a result, Algorithm 9 computes the same value of y as the original computation in lines 21–26 of Algorithm 2.

3.4 RMME-Based Evaluation of L in Signing

During the Signing phase, the construction of the matrices in L is shown in lines 3–4 of Algorithm 4. In the computation of $(P^{(1)} + P^{(1)\top})$, matrix $P^{(1)}$ resides in global memory, while matrix O is cached in shared memory, which makes this computation pattern well suited to an RMME formulation that reuses elements of $P^{(1)}$. By reusing the elements of $P^{(1)}$, this design significantly reduces redundant accesses to global memory. Since $P^{(1)}$ is an upper triangular matrix, the required elements are selected from either $P^{(1)}$ or $P^{(1)\top}$ according to the index coordinates during the computation. Once RMME produces the corresponding row result, it is directly accumulated into the corresponding row of $P^{(2)}$.

3.5 RMME-Based Evaluation of M in Signing

During the Signing phase, the construction of the matrices in M is shown in lines 11–13 of Algorithm 2. In this stage, the multiplication of a sequence of vectors by a matrix can be reformulated as a matrix–matrix multiplication. Since matrix L resides in global memory while the corresponding vector sequence v is cached in shared memory, this computation pattern is well suited to an RMME formulation that reuses elements of L. By reducing the number of accesses to L from k to a single pass, the proposed design significantly decreases global memory traffic. Once RMME produces the corresponding column result, it is directly written into matrix M.

3.6 RMME and Horner-Based Evaluation of y in Verification

During Verification, the computation of the intermediate value y can be viewed as an extension of the y computation in the Signing phase, as shown in lines 7–12 of Algorithm 3. Unlike Signing, the bilinear terms used to construct the intermediate values in Verification are no longer derived from a single upper triangular matrix $P^{(1)}$, but from a larger matrix structure composed of $P^{(1)}$, $P^{(2)}$, and $P^{(3)}$. Accordingly, for each index pair (i, j), the intermediate value can be expressed as

$$u_{i,j} = u_{i,j}^{(1)} \oplus u_{i,j}^{(2)} \oplus u_{i,j}^{(3)},$$

where the terms $u_{i,j}^{(1)}$, $u_{i,j}^{(2)}$, and $u_{i,j}^{(3)}$ are obtained by multiplying the corresponding vector sequences with matrices $P^{(1)}$, $P^{(2)}$, and $P^{(3)}$, respectively.

Since these three types of terms are mathematically independent and each follows the same computation pattern as in the Signing phase evaluation of y, the same restructuring strategy can be applied. $u_{i,j}^{(1)}$, $u_{i,j}^{(2)}$, and $u_{i,j}^{(3)}$ are computed separately and then combined through an additional accumulation step. In this way, the algorithmic semantics are preserved, while repeated accesses to large matrices inside the recursion loop are effectively avoided.

4 Design Overview

Achieving high-throughput MAYO signature computation on GPU requires not only operator-level optimization, but also systematic design choices in kernel organization and runtime execution. This section presents the overall design framework of the proposed GPU implementation. It covers kernel partitioning and fusion strategies, warp-level configuration selection, task scheduling mechanisms, memory pool management for intermediate data reuse, and asynchronous multi-stream execution for overlapping computation and data transfers.

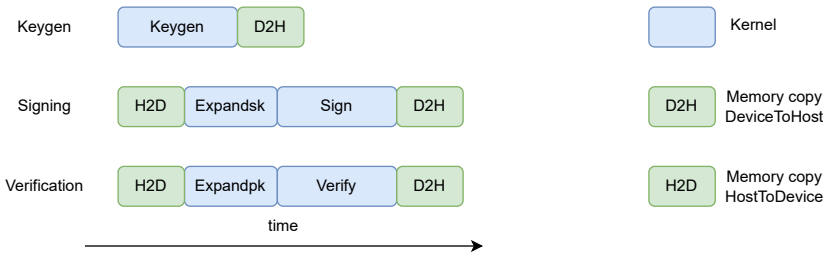


Figure 4: Kernel decomposition and memory transfer flow for Keygen, Signing, and Verification.

4.1 Kernel Partitioning

In GPU programs, frequently launching a large number of fine-grained kernels introduces significant launch and scheduling overhead. In high-throughput scenarios, this overhead can become a major factor limiting overall performance. Therefore, we adopt a kernel fusion strategy. The computations in the Keygen phase are fully fused into a single kernel. The Signing phase is divided into two kernels, Expandsk and Sign, according to the MAYO interface specification, corresponding to secret key expansion and the actual signing computation, respectively. Similarly, the Verification phase is divided into two kernels, Expandpk and Verify, as shown in Figure 4. This partitioning strategy satisfies the algorithm interface requirements while keeping the number of kernels to the minimum necessary, thereby effectively reducing the overhead caused by kernel launch and scheduling.

In addition, kernel fusion allows intermediate results within the same computation stage to be kept in registers or shared memory and passed directly within a kernel, without frequent write-back to and reload from global memory. This design significantly reduces the number of global memory accesses, further lowering memory access latency and improving overall execution efficiency.

4.2 Warp Configuration Selection

Different computation stages exhibit clear differences in computation structure, register requirements, and synchronization dependencies. Based on these characteristics, this work adopts differentiated warp-level thread block configurations in the overall design.

For the Keygen, Expandsk, Sign, and Verify stages, the computation flow has strong sequential behavior and state dependencies, and involves a large number of intermediate variables, resulting in high register demand. For this type of workload, we use a thread block configuration centered on a single warp, where the threads within the warp cooperatively complete the computation of a single MAYO instance. This design avoids splitting a single task across multiple warps, which would otherwise introduce cross-warp synchronization and scheduling overhead, allowing the computation to proceed sequentially with low synchronization cost.

In contrast, the Expandpk stage has lower register pressure and fewer synchronization dependencies among threads. For this stage, we adopt a thread block configuration containing multiple warps to increase intra-task parallelism and more fully utilize the computational resources of the GPU.

4.3 Task Scheduling

In task scheduling, the Keygen, Signing, and Verification processes are each divided into multiple relatively independent computation tasks and are executed in parallel with thread

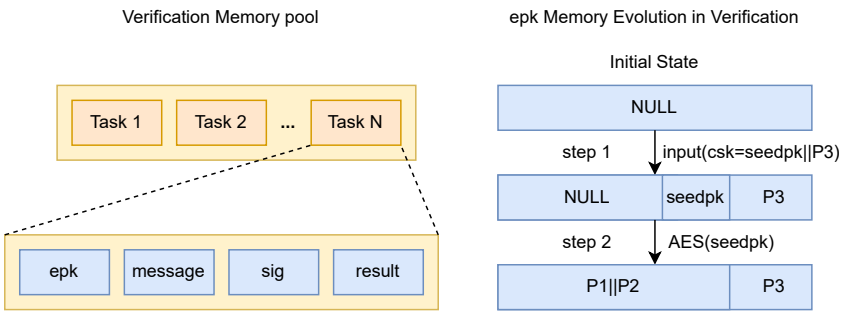


Figure 5: Memory pool organization and storage data reuse for the *epk*-related computation in the Verification phase.

blocks serving as the scheduling unit. A single kernel launches a large number of thread blocks simultaneously to process the same type of task in parallel, thereby supporting concurrent execution of multiple MAYO instances.

4.4 Memory Pool

To enable efficient memory access during concurrent task execution, this work adopts a task-level memory pool to manage the storage required by different computation stages in a unified manner.

This mechanism is optimized around three design aspects: memory reuse, data storage order planning, and alignment strategy. First, a memory reuse strategy is applied to intermediate data whose lifetimes do not overlap across computation stages. When an intermediate result is no longer accessed in subsequent computation, its storage space is directly reused to store newly generated data, thereby avoiding additional memory allocation. Second, the storage order of data in each stage is carefully planned to minimize the overhead introduced by data concatenation. Third, a strided allocation strategy is used to achieve block alignment. Considering that the GPU L2 cache operates with a cache line size of 128 bytes, bitstreams that are accessed non-contiguously are aligned to 128-byte boundaries to improve access patterns to L2 cache lines. The memory pool design for the Verification stage, as well as the memory reuse and storage order planning for *epk*, are illustrated in Figure 5.

By integrating the above memory pool mechanisms, the system effectively reduces the overhead caused by frequent memory allocation and deallocation, lowers the overall memory footprint, and efficiently supports large-scale parallel task execution.

4.5 Asynchronous Execution and Streaming

Asynchronous execution allows different computation tasks to proceed concurrently, enabling temporal overlap between computation and data transfer, thereby mitigating the impact of data transfer latency on the overall execution flow.

To more effectively utilize computational resources, we employ multiple CUDA streams to asynchronously schedule data transfers between the CPU and GPU alongside kernel execution. This approach enables concurrent progress of CPU-side tasks and GPU-side computation and helps hide data transfer latency to some extent, thereby improving overall execution efficiency and resource utilization.

5 Implementation Details

This section presents the implementation details of the proposed GPU-based MAYO scheme at the operator level. It describes the data representation strategy and the implementation of key computational modules, including AES-128-CTR, Gaussian elimination in SampleSolution, and the SHAKE128/SHAKE256 hash function.

5.1 Nibble-Sliced Data Representation

For data storage, we adopt a nibble-sliced representation following the design described in [BCC+24]. In this representation, each $\mathbb{F}_q$ element is stored using a 4-bit nibble, such that a single byte contains two finite-field elements, located in the lower and upper four bits, respectively.

For sequences of matrices with identical computational workflows, such as $\{P_a^{(1)}\}_{a\in[m]}$, $\{P_a^{(2)}\}_{a\in[m]}$, and $\{P_a^{(3)}\}_{a\in[m]}$, the storage layout is organized in a position-major manner. Elements from different matrices within the same sequence, but at the same matrix position, are stored contiguously in memory and packed into adjacent bytes according to the nibble-sliced representation. The elements at the next matrix position are stored only after all elements at the current position across the sequence have been stored.

During computation, the 32 threads within a warp perform identical operations across the matrices in the same sequence. Each thread reads a single byte from memory, thereby simultaneously obtaining the corresponding elements from two matrices at the same position. Since memory accesses by threads within a warp are contiguous in the address space, this access pattern can typically be coalesced into a small number of memory transactions by the hardware, thereby reducing memory access overhead and improving overall execution efficiency.

5.2 AES Implementation

This work implements AES-128-CTR encryption using two execution configurations, namely one-warp and four-warps, and adopts a T-table-based approach. This method combines the SubBytes, ShiftRows, and MixColumns operations into a sequence of table lookups and linear operations, thereby reducing the number of instructions.

A classical T-table implementation maintains four lookup tables, denoted as T0, T1, T2, and T3. Considering the limited shared memory resources on the GPU, we adopt a more compact variant that retains only the T0 table. The lookup results equivalent to T1, T2, and T3 are constructed at runtime using byte-level rotation operations. This design preserves the algorithm semantics while reducing storage redundancy and improving throughput.

5.3 SampleSolution Implementation

For the computation of SampleSolution, the core task is to solve a system of linear equations over a finite field. Following the methodology described in [BCC+24], we employ Gaussian elimination over finite fields to solve this system.

During the forward elimination and back substitution phases, we use warp-level shuffle primitives to broadcast the required scalar values within a single warp, thereby reducing the overhead of memory accesses and synchronization. In addition, the lookup table of finite-field multiplicative inverses required by Gaussian elimination is constructed and cached before computation, thereby reducing repeated computation during execution.

5.4 Hash Function

We adopt warp-level primitives and implement the SHAKE128/SHAKE256 hash algorithm following the method described in [SYD+24, OBS21]. During the state permutation phase, 25 threads cooperatively perform the 24 rounds of permutation operations, and warp-level shuffle instructions are used to enable cross-thread data access and updates. This design avoids explicit synchronization overhead.

5.5 Side-Channel Considerations

This work focuses on high-throughput GPU implementation. Therefore, we only provide a brief discussion of the main implementation-related side-channel considerations.

The RMME optimization itself does not introduce secret-dependent control flow or secret-dependent matrix-memory access patterns. Its loop bounds and matrix-access indices are determined by public parameters and fixed loop variables. Therefore, RMME changes the deterministic evaluation order and improves data reuse, but does not introduce secret-dependent branch divergence or secret-dependent matrix traversal. Consequently, the RMME optimization does not introduce side channels that could exploit secret-dependent execution flow or secret-dependent memory-access patterns.

Other side-channel concerns mainly come from table-based operations and shared GPU resources. The implementation uses $\mathbb{F}_{16}$ lookup tables and AES T-table/S-box accesses, which may lead to data-dependent memory-access behavior. Under stronger GPU attack models, such table-based accesses may be observed through cache behavior [ZCG+24], shared memory bank conflicts [LMZ19], memory coalescing timing effects [LMZ19], or co-resident workload attacks [NNQA18]. Developing effective countermeasures against these side-channel risks is left for future work.

6 Experimental Evaluation

6.1 Experimental Setup

We evaluate the proposed implementation on three representative GPUs with different computing capabilities, including NVIDIA Tesla A100, RTX 4090, and RTX 3090. The systems run Linux with CUDA support. In addition, the data transfer latency between the host and the device is consistently included in the overall measurements.

6.2 Cross-Platform Performance Comparison of MAYO

For the GPU evaluation, all performance results are reported as the average of 10 independent runs. In each run, a batch of 20,000 tasks is processed using 10 concurrent CUDA streams.

For performance comparison, we evaluate the proposed GPU implementation against existing MAYO implementations on different platforms, including reference CPU, optimized CPU, AVX2-optimized CPU, ARM microcontrollers, and ASIC. The CPU and AVX2 results are taken from implementations evaluated on an Intel Xeon Gold 6338 processor (Ice Lake architecture) operating at 2.0 GHz. The ARM results are obtained from an ARM Cortex-M4 implementation running on the STM32 NUCLEO-L4R5ZI development board, which is equipped with an STM32L4R5ZI Cortex-M4 processor, on-chip flash memory, and SRAM. The ASIC results are obtained from a specialized 28 nm implementation clocked at 1.5 GHz. The ASIC implementation in [HSK+24] was evaluated using an earlier MAYO parameter set. The MAYO parameter sets were subsequently revised by the designers during the standardization process, resulting in larger dimensions than those used in [HSK+24]. Therefore, the reported throughput results, including the MAYO_5

Table 3: Performance comparison of MAYO (ops/s). Ref/CPU/AVX2/ARM results from [BCC+25]. ASIC* results from [HSK+24].

Name	Level	Op	Our work(GPU)			Related works				
			A100	4090	3090	Ref	CPU	AVX2	ARM	ASIC*
$MAYO_1$	1	Keygen	308,238	680,383	293,790	400	2,276	16,849	2	258,665
		Signing	92,377	218,919	94,507	222	734	4,246	2	48,188
		Verification	207,241	423,533	186,777	482	5,655	13,049	2	327,011
$MAYO_2$	1	Keygen	313,584	717,689	303,904	428	2,579	20,771	2	-
		Signing	175,893	439,760	163,354	316	1,297	6,992	2	-
		Verification	546,498	1,170,878	523,059	752	19,177	35,477	3	-
$MAYO_3$	3	Keygen	140,139	278,565	126,513	148	574	7,081	1	127,877
		Signing	44,333	87,494	38,875	79	250	1,966	1	22,840
		Verification	84,832	181,439	80,474	196	2,280	5,748	1	166,021
$MAYO_5$	5	Keygen	43,908	93,109	43,328	62	283	2,609	-	75,388
		Signing	13,937	26,150	11,685	33	96	838	-	13,424
		Verification	34,279	58,152	26,847	93	1,126	2,342	-	101,950

* The reported ASIC results correspond to an earlier parameter set of MAYO. Consequently, the throughput figures are not directly comparable to the GPU results presented in this paper.

Table 4: Execution time per operation (ms)

Name	Level	Op	A100	4090	3090
$MAYO_1$	1	Keygen	2.172	1.342	1.832
		Signing	7.480	4.594	6.256
		Verification	2.931	1.695	2.308
$MAYO_2$	1	Keygen	1.572	0.808	1.116
		Signing	5.805	3.225	4.453
		Verification	0.989	0.671	0.887
$MAYO_3$	3	Keygen	4.961	2.799	3.994
		Signing	16.408	9.501	13.588
		Verification	6.186	3.620	5.171
$MAYO_5$	5	Keygen	13.653	7.379	11.126
		Signing	47.459	26.835	41.704
		Verification	15.699	9.305	14.216

Verification result, are not directly comparable to our GPU results and are provided only as a cross-platform reference. All reported performance results are converted to throughput metrics (operations per second) for Keygen, Signing, and Verification.

As shown in Table 3, on the RTX 4090 platform, compared with the CPU reference implementation, the performance improves by 1502–1882× for Keygen, 792–1392× for Signing, and 625–1557× for Verification. Compared with the AVX2-based implementation, the performance improves by about 35–40× for Keygen, 31–63× for Signing, and 25–33× for Verification. Unless otherwise stated, the following detailed analyses use the RTX 4090 platform, which consistently delivers the best performance among the evaluated GPUs.

6.3 Latency

Table 4 reports the single-operation latency of the proposed GPU implementation. Latency is measured by repeating each operation 10,000 times and averaging the results, and is reported in milliseconds. The reported latency includes the data transfer overhead between the host and the device.

We evaluate how the latency and throughput of the complete Keygen, Signing, and

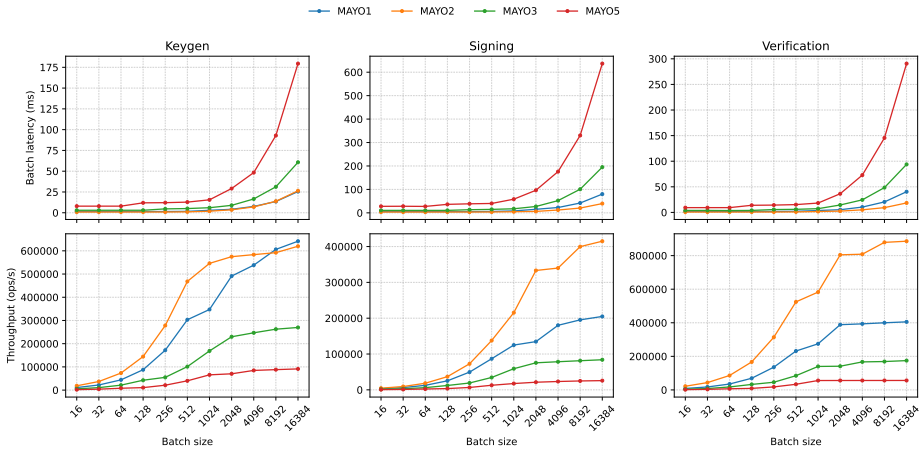
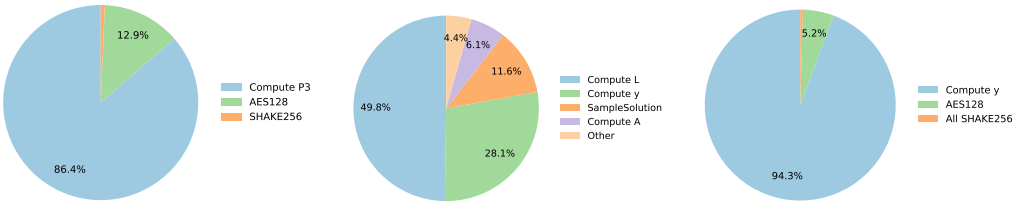


Figure 6: Latency and throughput under different batch sizes.



(a) Keygen (b) Signing (c) Verification

Figure 7: Latency breakdown of the proposed GPU implementation: (a) Keygen, (b) Signing, and (c) Verification.

Verification workflows change with the batch size using a single CUDA stream. As shown in Figure 6, before the GPU is saturated, increasing the batch size improves throughput while the batch completion latency changes only slightly. After saturation, the batch completion latency grows approximately linearly with the batch size, while throughput gradually stabilizes.

For the computation-stage breakdown, we use MAYO_1 as a representative example. As shown in Figure 7, the computation latency is decomposed into several key stages. For Keygen, the computation of $P^{(3)}$ dominates the computation latency, accounting for 86.4%. In the Signing phase, the main time consumption comes from the computation of L at 49.8% and the computation of y at 28.1%, which together account for 77.84% of the Signing computation time. In the Verification phase, the computation of y almost entirely dominates the cost, accounting for 94.3%. Overall, these results indicate that latency is primarily dominated by matrix computations.

6.4 Register Pressure and Shared-Memory Usage

We analyze the register and shared memory usage of the five kernels under MAYO_1 and MAYO_5. Table 5 summarizes the resource usage reported by the compiler. All kernels incur zero spill stores and zero spill loads under both parameter sets, indicating that the register allocation remains within the available hardware resources and does not cause register spilling.

Table 5: Register and shared-memory usage under MAYO_1 and MAYO_5.

Parameter	Kernel	Regs/Thread	Shared Mem/Block (B)	Spill (B)
MAYO_1	Keygen	80	2264	0
	Expandsk	78	2184	0
	Sign	128	6388	0
	Expandpk	78	1472	0
	Verify	128	4700	0
MAYO_5	Keygen	164	2820	0
	Expandsk	78	2724	0
	Sign	167	17619	0
	Expandpk	78	1472	0
	Verify	168	10579	0

Table 6: SM and DRAM throughput of different kernels under MAYO_1.

Kernel	Block Size	SM Throughput (%)	DRAM Throughput (%)
Keygen	1 warp	94.95	27.86
Expandsk	1 warp	88.13	27.17
Sign	1 warp	83.86	10.30
Expandpk	4 warps	77.47	25.96
Verify	1 warp	96.50	13.45

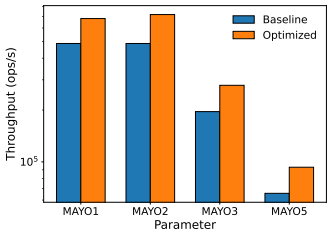
6.5 Performance Utilization Analysis

We report steady-state utilization under large-batch execution. Taking MAYO_1 as a representative example, as shown in Table 6, all stages achieve high SM throughput (77%–97%) while DRAM throughput remains below 30%, indicating effective on-chip data reuse and confirming that the selected thread-block configurations do not limit performance.

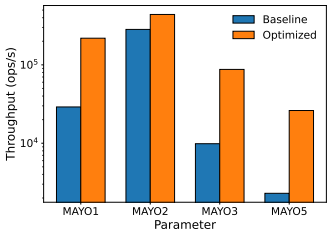
6.6 Effectiveness Evaluation of Algorithmic Restructuring

To evaluate the effectiveness of the proposed RMME-based optimizations, we perform a single-point ablation study. Each RMME formulation is enabled in isolation and evaluated at the computation stage where it is applied. The baseline implementation follows the original algorithmic evaluation order and employs the conventional inner-product formulation. Figure 8 summarizes the throughput speedup achieved by the five RMME-based optimizations across different MAYO parameter sets. All five optimizations improve throughput. The most significant gain is achieved by the RMME- and Horner-based evaluation of y in Signing, which achieves a speedup of up to 11.39×. The same optimization in Verification achieves up to 2.72×, while the remaining RMME-based optimizations also yield clear performance gains.

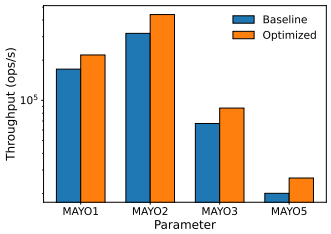
To further separate the contributions of RMME and Horner evaluation, we conduct an additional ablation study that isolates the y-computation in Signing under the same batch and stream configuration. Horner evaluation alone achieves 1.02× speedup, RMME alone achieves 18.90× speedup, and enabling both achieves 20.99× speedup. These results indicate that RMME accounts for most of the performance gain, while Horner evaluation provides an additional benefit when combined with RMME.



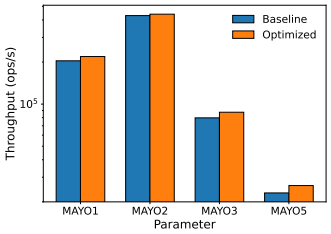
(a) Throughput comparison between the RMME based evaluation of $P^{(1)}$ and the conventional inner-product formulation during the Keygen phase.



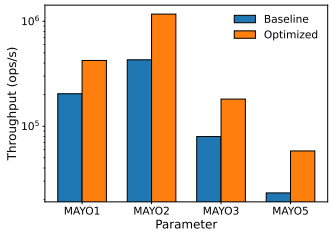
(b) Throughput comparison between the RMME and Horner based evaluation of y and the conventional inner-product formulation during the Signing phase.



(c) Throughput comparison between the RMME based evaluation of L and the conventional inner-product formulation during the Signing phase.



(d) Throughput comparison between the RMME based evaluation of M and the conventional inner-product formulation during the Signing phase.



(e) Throughput comparison between the RMME and Horner based evaluation of y and the conventional inner-product formulation during the Verification phase.

Figure 8: Ablation study of RMME-based optimizations across different computation components and stages.

7 Conclusion

In this work, we introduced a data reuse-aware computation model for matrix-dominated and memory-access-bound workloads. Building on this model, we proposed RMME as a computation reordering method for matrix multiplication to reduce redundant global memory accesses through operand-level data reuse. Building upon this method, we further developed a GPU-oriented hardware–software co-design framework that combines algorithmic restructuring, warp-level kernel design, task-level memory reuse, and asynchronous multi-stream execution to maximize end-to-end throughput.

We implemented the proposed design for MAYO across all parameter sets and evaluated it on multiple GPU platforms. Experimental results show that the proposed approach achieves orders-of-magnitude throughput improvements over existing CPU-based implementations. On the RTX 4090 platform, Signing and Verification throughput improve by more than two orders of magnitude over the CPU reference implementation, while Keygen throughput improves by nearly three orders of magnitude. In the best case, the implementation supports millions of verifications per second. To the best of our knowledge, this work achieves the highest reported throughput for MAYO_1, MAYO_2, and MAYO_3 among existing implementations.

These results demonstrate that RMME, combined with GPU-oriented co-design, is highly effective for accelerating multivariate polynomial-based cryptographic schemes. As future work, the proposed optimization principles can be further extended to other

multivariate polynomial-based cryptographic schemes.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China under Grant No. 62372417, in part by the Scientific Research Innovation Capability Support Project for Young Faculty under Grant No. SRICSPYF-BS2025137, in part by the Guangdong Provincial Key Laboratory of IRADS under Grant No. 2022B1212010006, in part by the Guangdong and Hong Kong Universities "1+1+1" Joint Research Collaboration Scheme under Grant No. 2025A0505000001, in part by the Guangdong Basic and Applied Basic Research Foundation under Grant No. 2024A1515011274, in part by the Singapore Ministry of Education Academic Research Fund Tier 1 under Proposal ID 24-SIS-SMU-052, in part by the Innovation and Technology Fund under Grant No. ITS/109/24, and in part by CityUHK under Grant No. 9440448.

References

[AAC+22] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status report on the third round of the NIST post-quantum cryptography standardization process. NIST Interagency/Internal Report 8413, National Institute of Standards and Technology, 2022.

[BCC+24] Ward Beullens, Fabio Campos, Sofía Celi, Basil Hess, and Matthias J. Kannwischer. Nibbling MAYO: optimized implementations for AVX2 and cortex-m4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(2):252–275, 2024.

[BCC+25] Ward Beullens, Fabio Campos, Sofía Celi, Basil Hess, and Matthias J. Kannwischer. Mayo round 2 version. Technical report, NIST Post-Quantum Cryptography Standardization Project, 2025. Latest update: February 5, 2025.

[BCD+25] Ward Beullens, Ming-Shing Chen, Jintai Ding, Boru Gong, Matthias J. Kannwischer, Jacques Patarin, Bo-Yuan Peng, Dieter Schmidt, Cheng-Jhih Shih, Chengdong Tao, and Bo-Yin Yang. UOV: Unbalanced oil and vinegar – algorithm specifications and supporting documentation. Technical report, NIST PQC Standardization of Additional Digital Signature Scheme, 2025. Version 2.0, February 5, 2025.

[Beu21] Ward Beullens. MAYO: practical post-quantum signatures from oil-and-vinegar maps. In Riham AlTawy and Andreas Hülsing, editors, *Selected Areas in Cryptography - 28th International Conference, SAC 2021, Virtual Event, September 29 - October 1, 2021, Revised Selected Papers*, volume 13203 of *Lecture Notes in Computer Science*, pages 355–376. Springer, 2021.

[BFST03] C Sidney Burrus, James W Fox, Gary A Sitton, and Sven Treitel. Horner's method for evaluating and deflating polynomials. *DSP Software Notes, Rice University, Nov*, 26, 2003.

[FIH+23] Hiroki Furue, Yasuhiko Ikematsu, Fumitaka Hoshino, Tsuyoshi Takagi, Kan Yasuda, Toshiyuki Miyazawa, Tsunekazu Saito, and Akira Nagai. QR-UOV. Technical report, NIST PQC Standardization of Additional Digital Signature Scheme, 2023. Specification document.

[FXL+22] Zonghao Feng, Qipeng Xie, Qiong Luo, Yujie Chen, Haoxuan Li, Huizhong Li, and Qiang Yan. Accelerating elliptic curve digital signature algorithms on gpus. In Felix Wolf, Sameer Shende, Candace Culhane, Sadaf R. Alam, and Heike Jagode, editors, *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, November 13-18, 2022*, pages 27:1–27:13. IEEE, 2022.

[GMSS23] Arianna Gringiani, Alessio Meneghetti, Edoardo Signorini, and Ruggero Susella. MAYO: optimized implementation with revised parameters for armv7-m. *IACR Cryptol. ePrint Arch.*, 2023:540, 2023.

[HSK+24] Florian Hirner, Michael Streibl, Florian Krieger, Ahmet Can Mert, and Sujoy Sinha Roy. Whipping the multivariate-based MAYO signature scheme using hardware platforms. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, pages 3421–3435. ACM, 2024.

[JMSS18] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. Dissecting the NVIDIA volta GPU architecture via microbenchmarking. Technical report, Citadel, 2018. arXiv:1804.06826.

[KPG99] Aviad Kipnis, Jacques Patarin, and Louis Goubin. Unbalanced oil and vinegar signature schemes. In Jacques Stern, editor, *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 206–222. Springer, 1999.

[LMZ19] Zhen Lin, Utkarsh Mathur, and Huiyang Zhou. Scatter-and-gather revisited: High-performance side-channel-resistant AES on gpus. In Adwait Jog and Onur Kayiran, editors, *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs, GPGPU@ASPLOS 2019, Providence, RI, USA, April 13, 2019*, pages 2–11. ACM, 2019.

[LSKY19] Sokjoon Lee, Hwajeong Seo, Hyeokchan Kwon, and Hyunsoo Yoon. Hybrid approach of parallel implementation on CPU-GPU for high-speed ECDSA verification. *J. Supercomput.*, 75(8):4329–4349, 2019.

[LWS+26] Wenqian Li, Hanyu Wei, Shiyu Shen, Hao Yang, Wangchen Dai, and Yunlei Zhao. cufalcon: An adaptive parallel GPU implementation for high-performance falcon acceleration. *IEEE Trans. Parallel Distributed Syst.*, 37(5):1153–1167, 2026.

[LZS+24] Wai-Kong Lee, Raymond K. Zhao, Ron Steinfeld, Amin Sakzad, and Seong Oun Hwang. High throughput lattice-based signatures on gpus: Comparing falcon and mitaka. *IEEE Trans. Parallel Distributed Syst.*, 35(4):675–692, 2024.

[NNQA18] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael B. Abu-Ghazaleh. Rendered insecure: GPU side channel attacks are practical. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 2139–2153. ACM, 2018.

[OBS21] Tatsuki Ono, Song Bian, and Takashi Sato. Automatic parallelism tuning for module learning with errors based post-quantum key exchanges on gpus. In *IEEE International Symposium on Circuits and Systems, ISCAS 2021, Daegu, South Korea, May 22-28, 2021*, pages 1–5. IEEE, 2021.

[ORCR20] Eduardo Ochoa-Jiménez, Luis Rivera-Zamarripa, Nareli Cruz-Cortés, and Francisco Rodríguez-Henríquez. Implementation of RSA signatures on GPU and CPU architectures. *IEEE Access*, 8:9928–9941, 2020.

[SA23] Seog Chung Seo and Sangwoo An. Parallel implementation of crystals-dilithium for effective signing and verification in autonomous driving environment. *ICT Express*, 9(1):100–105, 2023.

[Sho99] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Rev.*, 41(2):303–332, 1999.

[SYD+24] Shiyu Shen, Hao Yang, Wangchen Dai, Hong Zhang, Zhe Liu, and Yunlei Zhao. High-throughput GPU implementation of dilithium post-quantum digital signature. *IEEE Trans. Parallel Distributed Syst.*, 35(11):1964–1976, 2024.

[SYLZ25] Shiyu Shen, Hao Yang, Wenqian Li, and Yunlei Zhao. cuml-dsa: Optimized signing procedure and server-oriented GPU design for ML-DSA. *IEEE Trans. Dependable Secur. Comput.*, 22(3):2295–2307, 2025.

[SZM20] Shuzhou Sun, Rui Zhang, and Hui Ma. Efficient parallelism of post-quantum signature scheme SPHINCS. *IEEE Trans. Parallel Distributed Syst.*, 31(11):2542–2555, 2020.

[WCD+25] L. C. Wang, C. Y. Chou, Jintai Ding, Y. L. Kuan, J. A. Leegwater, M. S. Li, B. S. Tseng, P. E. Tseng, and C. C. Wang. SNOVA: Proposal for NIST PQC additional digital signature schemes. Technical report, NIST PQC Standardization of Additional Digital Signature Scheme, 2025. Version 2.0.

[WDC+25] Ziheng Wang, Xiaoshe Dong, Heng Chen, Yan Kang, and Qiang Wang. CUSPX: efficient GPU implementations of post-quantum signature sphincs$^+$. *IEEE Trans. Computers*, 74(1):15–28, 2025.

[ZCG+24] Zhenkai Zhang, Kunbei Cai, Yanan Guo, Fan Yao, and Xing Gao. Invalidate+compare: A timer-free GPU cache attack primitive. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.